Universitat de Lleida

TREBALL FINAL DE GRAU

Estudiant: Ramón de Llano Chamorro

Titulació: Grau en Enginyeria Informàtica

Títol de Treball Final de Grau: Artefacts of Power

Director/a: Francesc Sebé Feixas

Presentació
Mes: Septembre

Any: 2019

Contents

List of Figures

Glossary

In this section we introduce certain concepts that are necessary for the comprehension of the document.

- **RTS:** Real time strategy game. A genre of video games.
- **HP:** Health points. The integrity of a living entity.
- **Cast/Casted/Casting:** The process of charging a process during some time in order to perform an action after.
- **RayCast:** A built-in method from Unity which allows to detect collisions through a line[3].
- **Collider:** A script which performs computations in order to determine if two objects are having physical contact.
- **Vector2:** A representation of 2 dimensional vectors and points in Unity[5].
- **Vector3:** A representation of 3 dimensional vectors and points in Unity[6].

Chapter 1: Introduction

In this project we have developed a **real time strategy game**, or RTS, which includes at least all the minimum features to be considered. These features are: unit management, resource extraction and combat against other factions.
Not all the planned features could be implemented, but they have been totally planned and designed and they are explained in their corresponding chapter.

This project is about designing and implementing a real time strategy game using several tools. These tools are the game engine Unity, the 3D model editor Blender and several resources such as free sounds and sprites.

This project has been programmed in C# and using several Unity methods, the purpose of this report is explain how the most important features have been implemented, without entering deep into implementation details and focusing more in the algorithm and how to do it in a universal way.

Also, is important to mention that this project is focused on programming. This means that the employed models, textures, sounds and animations are minimalist and a lot of algorithms have been made even though the game engine includes them in order to have a better control, add or exclude certain features and prove the knowledge about video-games.

1.1 How to play the game

Though minimalist, this game is fully playable. Enemies will not actively try to win and attack the player, but the player has the absolute control of its entities.
In the following sections we explain how to play the game.

1.1.1 Controls

First of all it is necessary to know the controls and what the player can do. The controls themselves are easy since they are quite simplified, while the interactions between entities may be more complex. The interactions though are handled automatically depending on several factors.

Camera control

The player can look around moving the camera. The camera is moved by using the **arrows**, or if enabled, placing the cursor on the edges of the screen. The last method will move the camera in the direction of the edge where the cursor is placed. The camera can also be zoomed in and out using the **wheel scroll**. Lastly, the camera can be rotated pressing the **wheel click** and then dragging. To turn the camera to the original orientation, the player can simply click on the compass, on the edge of the minimap.

Unit selection

Units can be selected by **left clicking** on them. Additionally, several units can be selected using the area selection. To do so, the player has to press and hold **left click** in one point and **drag**. This will create a rectangle. Units and buildings inside the rectangle will be selected when the mouse button is released.

Unit management and interaction

To move units and make them interact with other entities, it is first required to have at least one selected. If the conditions are met, the player only needs to press **right click** on some other entity. Depending on the entity and the relationship with it, the unit or building will automatically interact with it in consequence. If no entity is under the mouse, the units will simply move to that position.

Unit abilities

Units and buildings may have abilities. Sometimes, the unit can use them by itself, while sometimes, the player has to trigger them.
Abilities can be triggered using the **numeric keys**. Certain abilities may be *binded* to other keys.

1.1.2 Objectives

As in many other RTS games, the objective is to improve and thrive in order to destroy the enemies. To achieve this goal the player has to do certain actions.

Obtain resources

The player needs resources in order to create and improve. Most resources are extracted from resource nodes, while some others can be acquired by other means.
To extract resources from a node, the player needs **workers**. To do it, the player has to select first one or more and make them interact with the desired resource. Workers extract resources over the time and they will actively find new ones unless the player stops them.

Spend resources

Resources are meant to be spent. The way a player spends them is decisive for the course of the game. Resources are basically spent in the following actions:

- **Create buildings:** Buildings are necessary to earn upgrades, increase the amount of trainable units and of course training them. Damaged buildings also need resources in order to be repaired.
- **Train units:** Units perform most of the actions and they are the most dynamic element in the game. Units are required for extracting resources and to fight the enemy.
- **Upgrades:** Upgrades are permanent, so they are usually worth to be obtained. Some of them even unlock better buildings and units.

Defeat the enemy

Any action has always the purpose of defeating the enemy. A player loses when he loses all its town centres. In order to destroy them, it is necessary to send units against them.

1.2 Design and implementation

1.2.1 Engine

The chosen engine has been Unity. The reason is because it is easy to use. The used language is C#, which may not be the best language to program a game of this kind, but is certainly fast and easy to use.
Unity offers a wide range of tools and methods really practical for game development. These methods have been used, but as explained before, some of them have been re-implemented or optimised.

Another reason to chose Unity has been the fact that it provides a free license and extension of its source code, making it really versatile and useful for projects of this kind.

1.2.2 Models

Blender is the software used to create the 3D models. It is also free software and very versatile.

As said before, models have not been developed in depth. Note that the models themselves are place-holders, but they constitute the base for a more elaborable ones. They are not lazy made, they just lack detail. They are the basis for future expansion, which would only need adding instead of modifying or subtracting.

1.2.3 Textures, particles and effects

For textures, we have chosen hand painting from blender. For the UI, most sprites have been hand painted using GIMP, another free software [2]. These sprites include all the UI icons and the map pointers.
For abilities, units and more elaborate sprites we have used other sources. Most units have been portrayed using sprites from other strategy games, while for the abilities we have used sprites found in Internet under free licenses.

About the particle effects, all of them have been hand made specifically for this game or taken from other owned projects, and all of them have been developed using the Unity particle systems.

1.2.4 Sounds

Sounds have been all obtained also from free repositories [1].

Chapter 2: Game manager

The game manager serves multiple purposes and is arguably one of the classes that performs more actions, but its main purpose is to read inputs from the player. It is the only class that uses the **Update** method from Unity, which is required to read key and mouse inputs.
Since it is not appropriate that a single class holds too many responsibilities, a lot of them have been delegated to other classes, which perform the actions triggered by the player inputs.
In the following sections we explain its main behaviours and functions.

2.1 Unit selection

Units, buildings and other entities can be selected, as explained in section 3. This section focuses on how they are selected.
As happens in any real time strategy game, it is possible to draw a rectangle on the screen in order to selected anything inside or clicking on a single one.

2.1.1 Single selection

In case the player releases without moving the cursor means that the player only wanted to select a single unit. In this case, the method **RayCast**[3] from Unity is used. This method detects objects that intersect with a described line. In this case, the line crosses through where the player clicked. The position of the screen is translated to the position in the game world, and a begin and end point are created. The x and z axis of the points are defined by the click point, but the y are constants and go from about the height of the camera to some meters below the ground. It is not necessary to check longer ranges since the described points are the maximum and minimum height an object could be found.

2.1.2 Area selection

Area selection allows the player to select several entities at the same time [7]. Area selection can be filtered, and select, for example, only friendly units (default) or, for example, select everything no matter if they are enemies or simple resource nodes.

This action begins when the player clicks. The input is read as **click down**. In this moment, the position where the player clicked in the screen is saved in a **Vector2**[5] variable.
If after clicking the player moves the cursor while holding the click (the position is different from the initial position) then a rectangle is drawn in the screen, symbolising the area of selection. This is made so that the player knows what he is selecting, but the rectangle is only a UI element.

This rectangle is drawn by calculating two opposite corners: the first is where the click down has been performed, and the second is the current position of the cursor. Then the 2D distance between both corners is divided by two: this gives a 2D middle point for the rectangle. The rectangle is then set in this point and scaled in each coordinate by the corresponding coordinate distance. Since the rectangle is created as size (1,1), it is easier to scale since the distance will always equal the axis length.

When the player releases the click is when the manager determines who is inside the rectangle. First of all, it is important to see that this rectangle in the screen can not be directly projected to the ground. The rectangle in the screen is a pure rectangle, while when translated to world-space coordinates, the rectangle becomes a **trapezoid**. This is due to the camera perspective, and since the angle is pointing forward, it means that the bottom of the rectangle is always wider than the top.

In the following figure it is explained the distortion effect. On the left, an in-game screenshot, showing the selection rectangle. On the right, the real projection on the ground of the same rectangle, seen from the ***editor*** *camera perspective. Note the deformation of the rectangle.*

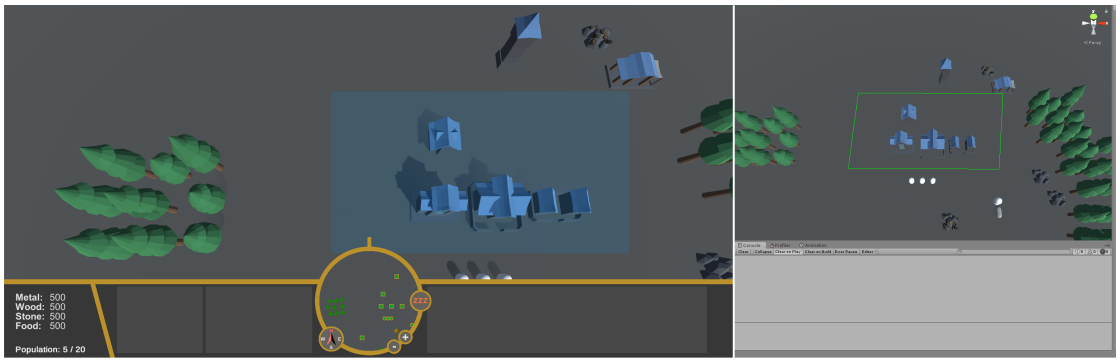

Figure 2.1: A visual demonstration of the camera perspective effect.

To project the real polygon on the map to know which entities are inside, first the other two corners of the rectangle are created by swapping the y axis while keeping the x axis (Note that this could also be done by swapping x instead). Once all the four corners are known, each one has to be translated to world-space positions. This is also made by **RayCasting** into the terrain.
Then, we know which are the true four corners in the game. Now, we detect which entities are inside this polygon. This is made the same way colliders work: checking object by object which ones are inside the polygon and which are not in.
The first step is decompose the polygon into minimum units: triangles. Since it is a 4 corner polygon, we only need to decompose it into 2 triangles. Each triangle has 3 corners. The chosen corners do not matter as long as they compose both triangles correctly.
Then, for each triangle we check if any entity is inside it. To avoid useless calculations, only entities (which are registered in a list) are considered, ignoring any other element such as decoration. Certain entities can also be ignored if the player wants it.
The selected entities are then added to a list, which is visually shown in the **entities panel** (Section 4.4).

2.2 Entities interaction

Entities can interact among them. Some interactions can be triggered automatically, but the player can always trigger them manually.
When an entity is selected and the player **right clicks** on another, the selected one will interact with the clicked one. The interaction depends on several factors, but mostly always there is only one possible interaction between two different entities.

The interaction range or position is given by the interacted entity to the interacting one using in most cases the circle-line intersection point [8].

- **Combat:** When having one or more selected entities capable of combating and clicking on another **live entity**, if the relationship (see Section 2.4.1) with this one is not friendly, the unit/s will engage combat.
  In case the entity can heal, and the clicked entity is a friendly **unit** and has been damaged, the selected one will try to heal the target one.
- **Gathering:** When having one or more selected **workers** and clicking on a **resource**, all the workers will begin to gather the selected resource.
- **Building entering:** Certain buildings can hold units. Most units can enter them if they are selected and sent to interact with them.

2.3 Camera

The camera allows the player to see the world. The camera is set above the ground, pointing at it with a forward inclination of 15º, which allows to have a better view and perspective of the world than a fully vertical one.
The camera is not static and can be controlled by the following actions:

- **Move with arrows:** the player can use the key arrows to move along the x-z axis.
- **Move with mouse:** it is also possible to move the camera using the mouse. When the pointer is on the edges of the screen, the camera will move to the corresponding direction. This feature may be inconvenient for some players and for that reason it can be activated and deactivated.
- **Rotate:** Using the mouse wheel click allows the player to rotate the camera around the y axis, which allows to have a view from other angles.
- **Zoom:** lastly, it is possible to zoom in and out. The values are clamped to avoid zooming too much or too less.

There is also another feature, the **camera focus**, which moves the camera to the position of an **entity** so that it appears in the middle of the screen. This transition is smoothed using a coroutine. This action can be performed by pressing F when having an entity selected, or by clicking on its **portrait** in the **selected entities panel**, which is explained Section 4.4.

2.4 Game rules and data

The game rules are different variables, constants and other values that define certain constrains of the game.

2.4.1 Players

A player is an object that represents a playing entity. Players (as object instances) contain a lot of information from the current game, such as their resources, their units, etc.

Relationships

A relationship is how a player interacts with another one. Relationships between players are chosen before the game begins, but it was planned that they could also be changed during the game development. This, in online mode, would allow to change the alliances between players depending on the situations.
Relationships are saved as two lists: allies and enemies. When interacting with a **live entity**, the relationship between the owner and the player who requests is checked.

A relationship defines whether a player can attack another one or not. The different relationships are:

- **Owned:** The entity belongs to the player. This is the strongest relationship and allows to control entities under this category.
- **Friendly:** The entity belongs to an allied player. Friendly entities cannot be controlled but they can be healed and cannot be attacked.
- **Enemy:** The entity belongs to an enemy player. Enemy entities will attack each other by default.
- **Neutral:** The entity belongs to a player whose diplomacy has not been defined (neither ally nor enemy) or to NPC. Neutral entities will not attack each other unless one of them starts the fight (will respond).

Resources

Each player has an amount of resources which are stored and saved in their corresponding instance. Resources, as in any RTS, are used to build, train and thrive. Players have to manage them properly.

All resources are extracted from resource nodes except the population. Those are explained deeply in the Section 3.1.
Population, on the other hand, is not extracted but gained by creating buildings that provide housing. This is explained in Section 3.5.6.

2.4.2 Rules

As said previously, rules are certain values that limit or define the game somehow. They can be split between dynamic and constant ones.

Dynamic

Dynamic rules are rules that players can choose before starting the game in order to improve their experience. Some examples are:

- **Resource yield multiplier:** it is possible to modify how many resources a node can yield before being exhausted. For example, trees could yield 150 units of wood, instead of 75.
- **Gathering speed:** modifies the rate at which a worker gathers resources.
- **Training speed:** modifies the speed at which units are trained.

Constants

Some rules are constant and cannot be changed by the players. These rules are intentionally forced to be like this to prevent possible issues. Some of them are:

- **Maximum amount of houses:** Its purpose is to prevent an overwhelming army, which could also cause performance issues, though its main purpose is to force the player to have a better management since it is a **limited resource**.

2.5 Player preferences

Player preferences are a collection of values that players can edit on their own. They are mostly focused on the UI. Some examples are:

- **Sensitivity:** Maybe it is the most important preference. It changes the speed at which the mouse moves, how fast a camera zooms or moves, etc.
- **Colours:** Players can choose in what colour they see, for example, the health bar of the enemies.
- **Unit selection:** The player can choose also that when **area selecting** entities, it ignores enemies, buildings, resource nodes, etc.

Chapter 3: Entities

> *Entity: Something that exists apart from other things, having its own independent existence*
>
> – Cambridge dictionary

Entities refer to any in-game element that is interactable somehow. This includes the units, the buildings, the constructions and the resource nodes. Entities share common properties: they all can be interacted, destroyed and targeted. They have attributes and values. That is why they receive the name *entity*. That excludes the rest of objects, which are basically environment and static.

3.1 Resource nodes

Resource nodes are entities which serve the purpose to be extracted by **workers** in order to yield **resources**. In the game there are four kinds of resource nodes while being five resources. That is because population is not extracted but created, as explained in 3.5.6.
Resource nodes are not owned by anyone and they do not have any attribute besides their integrity, which is the amount of remaining resource, though it is treated like HP from units and buildings. In the following subsections we explain each resource node as well as the resource itself.

3.1.1 Trees and wood

Trees are the resource nodes that yield wood (75 by default). Wood is mainly sued to build and repair, but some units also require it.
Trees are the most abundant resource. They appear in groups (forests). Sometimes, forests can also be used as a defense since they can cover certain angles, and they can be used as a natural wall.

3.1.2 Stone and quarries

Quarries are nodes that yield stone. Quarries are more limited than trees, but still more abundant than metal.
The stone is used for building and repairing along with wood. Very few special units may need it.
Quarries appear all over the map, but they can also spawn from an exhausted metal mine, as explained in the next subsection.

3.1.3 Metal mines and metal

Metal mines are the least abundant resource node, and thus in most of cases, the most important one. Metal mines yield a lot of iron over the time though. The idea of making them less abundant but yielding a lot is to make the players fight for them as they become strategic nodes by its properties and nature. Exhausted metal mines become quarries when all the metal has been extracted.
Iron is used mainly for units, though for some upgrades. This is what makes it so valuable.

The model used for ore encrusted in the mine, though resembles to gold, they are cubes of pyrite, a compound mineral formed by iron and sulphur.

3.1.4 Food

Food is a resource which purpose is to create units and some upgrades. Food is not extracted from only one node, but can be obtained from several:

- Bay bushes and fruit trees: they work exactly like the rest of resource nodes.
- Food yielding NPCs: they are live entities which, when killed, yield their corpse as a resource node so food can be gathered from them.
- Farms: Farms are buildings constructed by a player which when holding a worker inside, they generate food over the time. This is better explained in its corresponding section.

3.2 Live entities

Live entities is the name given to the entities that are owned by a player. They have health points and so they can be damaged by other players. Live entities are basically the units and the buildings. Also, live entities have the following base stats:

- Health points: the amount of live they have. When it reaches 0, they die.
- Defense: mitigates the received damage.
- Sight radius: the radius a unit can see, revealing other entities.

Additionally depending on the entity they can have other properties:

- Attack: affects to the dealt damage. Not all live entities have attack, like workers or houses. Note that some buildings do, such as the watchtowers.
- Movement speed: how fast a unit moves around the map. Movement speed is exclusive for units, and all units have it.

As said before, they can be attacked and damaged. If the health points reach 0, they die. When dying, they cancel any actions they were doing: attacking, gathering, training units, etc.

Units can be healed by healing spells or by resting close to a **town hall** and not in combat. This last feature adds a meaning for retreating, when in most strategy games units that are low health are pretty useless if they cannot be healed and most times they would be just sacrificed.

Buildings, on the other hand, cannot be healed as such, but they can be repaired. Only workers can repair buildings. That action spends some resources (basically wood and stone).

3.3 Units

Units are the entities able to move around and have several interactions.
Units are configurable from the editor, meaning that each one have different attributes such as skills, powers, uses and stats.
Units include workers, basic soldiers, vehicles such as tanks and even flying fighters.

3.3.1 Movement

As said before, units can move. The implemented movement is quite primitive and the unit is limited to go in euclidean distance to the goal position. First, the unit aims to the position using the Unity method **LookAt**, which rotates a game object towards a position. Then, the unit starts a coroutine, which moves the unit by its speed until reaching the goal position. This coroutine can be cancelled by manually ordering the unit to stop or by other in-game means. If the player changes the goal position while the unit is moving, the coroutine will be still executed but the unit will be aimed to the new position.

Knowing if the unit arrived to its destination requires to perform a loop. Every **fixed update**, two subtractions are computed in order to know the distance to the goal position and two equality checks to know if the distance to the goal is less or equal than an *epsilon* constant.
Calculating the y axis is unnecessary since units only move over the x-z plane, and lightens the computation by one third.
The named before *epsilon distance* is a small float which purpose is to avoid skipping the goal point: since we are not working with discrete values, the distance would most likely never be exactly 0, and an equality check of *distance == 0* is not possible. When over-passing the goal position, the distance will then start to increase and the unit will never stop. This epsilon is calculated by multiplying the speed of the entity by the time elapsed between this frame and the last. This is basically the length of a step in a frame.
Having a distance value below the epsilon value is enough to consider that the unit arrived to the destination. In the last execution of the movement loop, the unit is directly placed on the goal destination. That is always an smooth transition since the epsilon distance is always less than a unit step and will never look like the unit made a larger step or a "teleport".

At the beginning, the movement was performed by precalculating the distance, then interpolating between the beginning to the end, by t, which was a value that was increased over the time in function of the speed of the unit and the distance to be travelled. This method proved to be less efficient than just moving and checking every step if the unit reached the goal position, since interpolation more complex to calculate than the distance.

There is a special case where the unit does not move to a goal a position in the map, but towards another entity, specially a unit. In this case, the movement is not performed in the same way since the goal position is constantly changing. To solve this, the unit has to call the method **LookAt** to aim constantly to the target.

*Note that the method above could also be optimised in the following way: a unit A moves to a position. Unit B chases unit A. The unit B does not need rotate constantly to unit A, only to know where the unit A is heading and then intercept. In case the unit A changes its movement, it notifies unit B (We could use the **Observer pattern** in order to notify all the units that are targeting it) to recalculate the intersection position.*

3.3.2 Formations

Formations is the way a player can create groups of units to fight together in a certain arrangement. For example, placing heavy defensive units in front while keeping in the back lines powerful damage dealing units at range.

At the beginning it was planned that the player could edit them with an overlay on the UI. That would consist into showing the polygonal shapes (basically squares and triangles) and then allowing them to edit the amount of rows and columns as well as the spacing between units. Also, allowing them to place certain units further or closer to the center of the formation. This has been discarded since it may be over-complicated, so default shapes are provided.

Arrangement

The arrangement consists into shaping the formation. As said before, it can be done as a triangle or a rectangle, but the basic arrangement is a **line**. From that, the other rest of polygons are composed. If it is a triangle, first a line of one is created, then of two and so on. For a rectangle, the line is always the same size (except for the last, which would take the rest of remaining units). The process to decide the positions of a formation is the following:

- First, it has to be taken in mind the amount of units that compose a formation. For an square for example, the first thing to calculate is the **side**. This is calculated by computing a square root.
- The next step is to calculate the height of the formation. For a square it will equal the width. For a triangle, it is calculated by subtracting an accumulative value until the remaining amount of units without position is zero (first row is one, then two, then three, etc).
- Now it is the moment to create lines or rows. First of all, the width is split in half and negated. This will be the offset. The center of the formation is 0, so the units at left are in a negative position while the units at right are positive (taking the formation as the pivot). The offset is added to the x coordinate of every unit. For the height it is done exactly the same, but with the corresponding coordinate y (or z if we take in mind the position system of Unity). Each coordinate is also multiplied by the **spacing** between units.
- Now comes the last and hardest part which is to take in consideration the **rotation** of the formation. To do so, it is necessary to calculate the angle. The angle can be obtained by accessing the Unity GameObject field **EulerAngles**, but it has to be normalized to radians to operate it and offset it to match the Unity rotation orientation system.
  Once the angle is know and normalized, the last necessary thing is to multiply the x coordinate by the *cosine* of the angle and the y one by the **sine**.

3.4 Buildings

The buildings are owned entities that cannot move.
As well as units, buildings also have several attributes.
Some examples of buildings are the town center, houses, barracks, etc.

3.4.1 Town center

The town center or town hall is the core of the game and, of course, it is the most important building. It has several behaviours, working as a multipurpose building.
The player starts with one town hall at least, and losing all of them could imply losing the game (depending on the game rules).

Although not implemented, town halls would not be able to be placed freely and infinitely. Taking inspiration from other games, the appropriate would be to be restricted to certain, finite zones in the map.
In the current state of the game, the town hall is unique and you cannot build more.

Workers are created in town halls. Additionally, town halls can also attack like defense towers. Lastly, each town hall provides **housing**. By default, 15 units.

3.4.2 Houses

Houses are simple buildings that serve the only purpose of increasing the available population in order to train units. Each house provides 5 units, and houses are limited to 10 units.

3.4.3 Barracks and sorcery institute

Barracks is the building that allows the player to create basic, physical damage units. They have a moderate cost and are mainly used in the beginning of the game. On the other hand, the sorcery institute provides more complex and expensive units for ranged combat.
Using right click when having them selected will set the position where the newly created units will move when created.
They server as unit trainers. Their behaviour is explained in the following sections.

3.5 Live entities behaviours

Live entities are characterised by being able to perform certain actions, tasks and abilities.
Abilities are actions an entity can perform, and they are triggered manually by the player or automatically by the entity when given such privileges.
Tasks mainly refer to continuous actions. Tasks have no icon so that they cannot be triggered like abilities, but most of them are triggered manually by the player.
In the following subsections we explained certain behaviours and functionalities an entity can have.

3.5.1 Abilities

Abilities are the most complex part to develop for the game. The difficulty comes with the wide range of different effects, ways to use them, etc.
Several approaches have been made, but the last and most effective one has been to use the **Composition and Strategy pattern** in order to minimise the amount of code needed. This however caused a quite complex system though it requires very few additions in order to create a new ability, and has a lot of reuse.

First of all, it was necessary to think about all the possible abilities that could exist, extract each one of their individual behaviours and then encapsulate all of them in several categories.
All the abilities have, at least, the following attributes:

- **Name:** A unique name that identifies the ability.
- **Sprite:** An image that represents the ability in the UI.
- **Description:** A text field that explains what the ability does.

Also, every ability has a **range**, a way to be **used and triggered**, a way to be **delivered**, the **effects** it performs and the way to be **cast**.

Range

It determines how far the ability can be used taking in account that ability has to be used on a **target**. Otherwise, it is not necessary and should be *null*.
Note that the range is not a numeric field but an **object** which has a numeric field. Abilities use the reference to an object. The reason to use objects is because it allows to create some "constants". For example, the *long range ability* is set to 20 meters. If, at a given moment, is necessary to change the range of all the long range abilities, it is as easy as changing the value of a single object to affect all abilities. The same would go for melee abilities.

Costs

Costs are a list of necessary resources in order to use an ability. Although combat costs (mana, rage, energy, etc) have been removed for combat abilities, all **building placing** abilities require it. The costs though are **resources**, mainly wood and stone.
An ability will be disabled if the cost is greater than the available resources, thus preventing its triggering.

Pre-use mode

Pre-use mode or summon mode refers to the certain actions that a player has to perform before an ability can be used. Mostly all abilities are directly used, but there are several exceptions:

- **Building placing:** Placing buildings is considered an ability. To place a building, first it is required to choose a place. When clicked a valid position, the **construction** will appear.
- **Area of effect:** Area attacks that are not constrained to a target require a position in the world. The player not only sees the central point of the area, but can also see its effective area range. They work in a similar way as the building placer.

Cast mode

Cast mode refers to how the ability is performed and charged. Not all abilities are instantaneous, and some require time. The following are the modes an ability could be casted:

- **Instant:** The ability is delivered right after being triggered.
- **Cast charge:** Requires the entity to charge for certain amount of time. When the time is fulfilled, the ability is triggered.
- **Channelling:** The ability is performed periodically during the channelling time. This is opposite to charging. Channelling implies an amount of performs, or ticks, and a rate interval.

Deliver mode

All abilities are delivered somehow to the target. All melee attacks, for example, are **direct**, but ranged attacks are mostly always delivered via projectile. There are several ways to deliver the ability, which are encapsulated in an algorithm inside a class, implementing an interface **DeliverMode** (Strategy pattern). The current deliver modes are:

- **Direct to target:** Deliver the ability directly on the target. All melee attacks are delivered this way.
- **Deliver to self:** Delivers the ability directly on the **caster** or user. An example is **self-healing**.
- **Projectile:** The effect/effects are contained in a projectile, which delivers them on contact with a valid target. Note that projectiles ignore non-valid targets and go through them. For example, a fire ball would go through any friendly unit but would impact on any enemy or neutral entity, as well as any obstacle. In the same way, a healing projectile would go through enemy entities.
- **Area:** The effect is delivered to all or some entities inside an area. It can be a radius, an sphere[4], cone, etc.

Requisite

A requisite is a certain rule which has to be met in order to be able to use or just apply an effect. Some examples are:

- **Relationship:** It requires a certain relationship with the target. For example, it is not possible to heal enemy units or damage allies.
- **Owned state:** The caster requires a certain state. For example, the caster requires to be *burning* in order to cast a certain spell.
- **Target state:** The target requires to be under a certain state. For example, if the target is burning, a fire ball would deal double damage.
- **Terrain:** The ability requires to be cast on certain kind of terrain. For example, freezing the water or turning sand into a trap.

Effect

The effect is the most important component. It contains what the ability actually does. Effects is the component more prone to be extended, as new abilities appear, but the basic and implemented ones so far are:

- **Damage:** Deals direct raw damage to the target.
- **Healing:** Deals direct raw healing to the target.
- **Status effect:** Adds an status effect directly to the target.
- **Effect over the time:** Similar to an status effect, the effect over the time deals an **effect** every n seconds, for t times. Note that this has to be treated carefully since it can fall into an infinite recursion loop if not used correctly.

3.5.2 Combat

Combat is a general behaviour that allows an entity to fight against other live entities. Both buildings and units can have it.
Combat can be engaged manually by **ordering** an entity to attack another one, or can be triggered by **proximity** or **continuity**.
The efficiency in combat is determined by the entity stats and their abilities. Some units may be more effective against others.

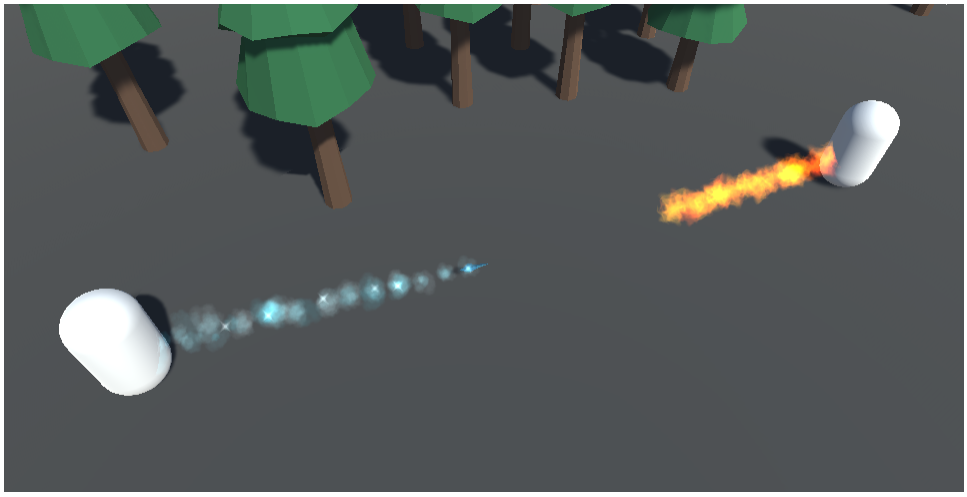

Figure 3.1: Two units (magic casters) fighting each other.

Engaging

Engaging in a combat may happen by the three means named next:

- **Ordering:** Consists into sending manually and directly an entity or more to combat another. This will change their target and focus to the new one. It will also cancel any task. Of course, ordering an attack requires to have at least one entity selected that is able to fight. Non fighting entities will ignore the order and will continue with their tasks, if any.
- **Proximity:** Units can also engage in combat if they are not set as passive and an enemy unit enters in their sight radius. In this case, units may or may not chase a unit if it flees depending on the **combat stance**, explained in the Section 3.5.2.
- **Continuity:** When an entity fights another and its target dies, it will search for a new target. Losing its current target (because it dies) triggers this action. The entity will try to find one in its radius of sight and if it does it will continue the combat. This process is repeated until either the attacking entity dies, all the enemy entities in sight die or disappear from the line of sight, or the player cancels the combat manually.

Enemy detection

As said before, units and buildings can see other entities depending on their sight radius. To do so, is necessary to scan their proximity. We have implemented two methods for that, after the first one proved not to be flexible.

*The first approach to detect when a unit enters in the field of view of another was to use a passive call reception, instead of using an active **collider** or **trigger** for each unit. It was not the unit that was static the one who detected the unit entering in its field, but the unit that moved the one that notified the static one that it entered in its field. This was made this way to avoid constant collision checks and reduce the load.*
The problem came when entities started to have different sight radius. A unit with a longer sight radius could notify one that had less and thus was not in the field of view, and vice versa.

That is why we tried a new approach. It was also noted that it is pointless to check if a building is in the range of sight of another building: either it is from the beginning or never will be. The same applies to two distant units that are idle. Only when one moves there is a chance that they see each other.
When a unit is moving to a position, it will not engage in combat if it sees another enemy unit or building unless this one attacks it first since its task is to move (and entities always try to fulfil their ordered task). But if it gets attacked first, the unit will react, so there is no problem to detect the combat here.
The real moment a unit has to detect combat is when it is idle. This way, units only search for combat against other units and buildings when they are idle (or in combat of course). When a unit arrives at its destination, it starts the coroutine **SearchForCombat** which instead of using a collider or trigger it uses the method **OverlapSphere**[4] every n seconds. The n value is set by default at 0.5, which proved to be performance friendly and a rate fast enough to always detect combat accurately.

Note that this method could also be optimized ignoring the y axis and creating a new method that only checks distance x-z and between live entities that are actively searching for combat. Units that move or are in passive stance could be removed from the list of entities.

Target choice

Right now, the way the target gets chosen automatically in combat is by proximity, though the system could be extended by adding the possibility to change the algorithm using **Strategy pattern**, and having different algorithms, allowing the player to choose the most suitable in each moment.
Some of the proposed extra algorithms are:

- **Target entity with lowest health:** helps to finish off units, creates more focus and ensures that entities will die in an engage instead of scattering damage and allowing the opponent to win and heal. As downside, it is not efficient damage/time, since it causes a lot of overkill: large amount of damage not necessary to kill an entity wasted since it can die with less.
- **Target entity with health below n:** this could help to mitigate the overkill of the algorithm explained above.
- **Target entities of certain class**: for example, focus on sorcerers and archers since they deal a lot of damage.

These algorithms are not limited to only combat continuity or proximity but they could also be used for engages.

Combat stance

The combat stance is how a unit or building behaves against a combat engage. There are several combat stances and can be changed in any moment for units that are capable of fighting.
The combat stances are:

- **Aggressive:** The default stance. It will engage in combat against any enemy entity. It will also continue combat against any visible enemy entity.
- **Passive:** Passive entities do not attack other live entities, even if they are being damaged. Also, triggering this combat stance in the middle of a combat will cancel any cast and stop the combat.
- **Stand their ground:** Standing the ground is a limited aggressive stance. The unit will fight at first sight as aggressive, but will not chase a unit farther than a certain distance. The distance is defined by a constant and cannot be changed by the player. In order to track the distance, the unit stores the last position where the unit has been sent.

Combat stances are implemented as abilities, and they can be triggered from the abilities panel (see Section 4.3).

3.5.3 Worker

Workers are the backbone of the game. They gather resources which can be later spent on buildings (which are also build by them) and units, upgrades, etc.
In the following subsections we explain how they perform their tasks.

Gathering

Gathering consists into extracting resources from **resources nodes**. Each worker can extract and take one resource type at a time while multiple workers can work on the same resource. Each time unit, a worker extracts 1 unit. The time needed to extract one unit depends on the upgrades. By default, 1 unit per second and 10 units to carry. A worker cannot keep 2 different resources at a time (below we explain what happens when changing the resource type).

When a worker is sent to interact with a resource, the worker will move to it. In case the worker is sent when already having a a resource, and the new resource is different, it will first go to store this one and then will go to gather the new one.
When reaching the resource node, it will start a coroutine. This coroutine extracts 1 resource per time unit. The worker will keep doing this until either the resource node is exhausted or the worker is full. When full, the worker searches for the closest storage entity. This is not limited by the sight radius, since the player has knowledge about its own entities.
Once the worker reaches the storage, it will store the resources, adding them to the global player resources. After that, the worker will automatically continue extracting the resource from exactly the same resource node.

When a worker exhausts a resource node, it will actively search for a resource of the same kind. If found, it will continue with the tasks. This is practical when extracting wood from trees, since they are small resource nodes and they get exhausted quickly.
In case a resource is not found, the worker will go to store the resource, and when done, it will become idle. This is made to avoid unwanted actions, such as wandering around the map just to gather a far metal mine after cutting down a tree.

Building

Building is the other fundamental task performed by a worker. The workers are the ones that have in their ability panel the different possible buildings to build by the player (this is explained in abilities section 3.5.1).
When a worker is sent to a construction, it will move to the interaction position and then it will start to increase its progress. This works no much different as with resource nodes. Also, when the worker finishes a construction, it will actively try to find other constructions. In this case, the worker requires to have the construction in the radius of sight. When all visible constructions are finished, the worker will become idle.

Repairing

Workers can build, but also repair damaged buildings. Repairing a buildings works really similar to building: the worker goes to the interaction position and increases the progress, or in this case, the HP of the building. As well as with construction, it will search actively for near (in sight) damaged buildings to repair.
The difference here is that the worker consumes resources to repair (wood and stone). The amount required per unit work is defined in the **game rules** (section 2.4).

3.5.4 Unit trainer

A unit trainer is any **building** which is capable of training units. Units trainers are the town center, which produces workers, the barracks, which produce most of combat units, and the workshop, which produces heavy siege machinery.
Unit trainers have a training queue, and can accept several training tasks. The completion order is FIFO (first in first out).
The training cost of a unit is subtracted when it is added to the queue from the corresponding player resources.
If a player does not have enough available resources, the ability to train a unit will not appear available and thus it cannot be used. Note that in order to avoid spending non available resources it is also necessary to check, when a training is completed, that the player still has enough resources. That is because the player could spent resources during the training time.

3.5.5 Unit container

Unit containers are buildings or even other units which can keep other **units** inside them. Some units may not be able to enter certain buildings.
Unit containers have an skill that allows them to take out all the units contained in them. Units on the other hand only need to interact with the unit container to enter. The amount of units that can be contained is limited and variable.

Some examples are: the watchtowers, the town centres and the steam tanks. The last one acts as also as a kind of transport.

A beta feature is that certain unit containers would use different combat abilities depending on the unit contained in them. For example, a watchtower would throw fireballs instead of arrows if the unit inside is a mage. That was also the main feature of the ***steam tank****.*

3.5.6 Housing

Housing refers to any building that provides an increment of trainable units amount. **Houses** server this sole purpose, while they are not the only ones that can provide housing.
Town centres also do (by default 3 times more) though they are way more hard to build and more limited.
When created, a building that provides housing, they update automatically the available population for the player. When destroyed, they subtract the total amount of available population.

Note that population is a **resource**, and it is implemented in the same way as wood for example, though is manipulated differently in some situations.
Population is only used to create **units**.

3.5.7 Resting area

A resting area is an area, around a **building**, where friendly units that are not in combat are healed over the time (by default, 10 HP each 2 seconds). This behaviour is, by now, exclusive for **town centres**.

*Originally, this behaviour was supposed to be used by a specific buildings: the **Inn**. At the end, it was considered that the town center would fulfil this purpose better. The reason is because a player would tactically place inns around the map as a resting area, for example close to a fight, in order to get a quick a healing source.*
The reason to implement this behaviour is to incentive players to retreat and not sacrifice units. Town centres force the player to have only specific, predefined areas for that.

3.5.8 Farm

A farm is a building that produces food over the time whenever a worker is inside it. The farm behaviour is exclusive for farm buildings, and they also have the **unit container** behaviour.
Food then becomes the only resource that is unlimited in the game.

Chapter 4: Interface

The interface allows the player to manage its resources. The interface has a real time map, the selected units panel, a portrait for the focused entity displaying all its properties and more.

4.1 Resources

The resources of the player accumulated during the game are shown in the left corner. The amounts are updated automatically whenever there is a change on them. They are also highlighted when increased (green) or decreased (red) in order to be more visual an intuitive for the player.
The update calls are basically made by the **storages** whenever a worker stores resources or different buildings use them such as **unit trainers** and **upgrades**. No Update method is used, so there is no constant polling.
On the same way, every time the resources are updated the mentioned structures are notified, and units or upgrades that could not be possible to do due to lack of resources are set as not available.

4.2 Portrait

The portrait is the element of the UI that shows all the properties of an entity. The properties shown vary, but mainly, and since it is focused to live entities, it shows the attack, defense, movement speed, sight radius, etc.
The portrait can only show one entity. The shown entity is the **focused** one. Focusing can be performed by clicking on an entity from the selected entities panel (explained in section 4.4) or when there is only one selected unit (default).

As well as in other cases, the values are not updated by the Update method. All the values are updated from the entities itself. Specially, by live entities, which update the value of their health integrity every time it changes. To do this, the live entity checks if the focused entity equals to itself.
Direct access to this is performed using a **Singleton**.

4.3 Entity abilities

The entity abilities is a panel that holds several icons representing the abilities an entity (the focused one) can perform. For combat units, it shows their combat abilities, for workers it shows the available buildings, and for buildings it shows the trainable units and upgrades. It does not matter how they work and what are they purposes, all abilities are treated the same way.

4.4 Selected entities

The selected entities is a panel that contains a portrait for every selected unit. This panel does not only allow to have a view on all the selected units, but it is also possible to change the focus by clicking on the corresponding portrait.

4.5 Mini map

The minimap is a key element of the UI. It shows the map from an up-view perspective and allows the player to see what is happening on its surroundings. In most real time strategy games, the map is set into a corner, while in the center remains, usually, the properties of the selected unit or building. The truth is, in most games a player gets used to what a unit or building does, and its properties are know beforehand since they have a constant value. On the other hand, the map is something dynamic and always different, and thus, it has been considered that it deserves a privileged place: the center. Hence, the map is set in the middle of the UI, not only making it more aesthetic, but also giving it the deserved importance.

The camera of the map is orthographic, meaning that the perspective is ignored and all the elements are displayed in 2D as the distance to the element is irrelevant.

The purpose of the map is to translate a position in the game to a UI representation. The map **does not render the units** as they appear in the field, but reads all units in the game, filters the ones that are in the radius of sight, and then creates a pointer in the UI referencing an entity relative to the combat field.
The mini-map uses a pure mathematical and discrete system to render the true position of the entities, instead the initial version, which was limited to render a 3D object attached above all entities and culled by the main camera.

Figure 4.1: The minimap showing different icons for different entities.

An entity in the map is pointed by using the following steps:

- The first step is to know the **distance** from the entity to the map. This is as easy as subtracting two positions: the minimap position to the entity. The positions of course are Vector3[6], but the y axis is unnecessary here since the map does not represent depth, so only the x and z axis are computed.
- The second step is normalise the value (Turning it into a float between 0 and 1). To do this, the built-in method **InverseLerp** is used, which takes the distance value and interpolates it into a relative value, considering the minimum value the negative radius of the map camera sight, and the positive radius.
  Note that since the camera is in orthographic mode, getting the radius is as easy as reading its value from the map camera size.
- Now that we obtained the normalised value, it is necessary to do the step two reversed for the UI map. The in-map x and z values are the interpolated value of t between the negative map radius and the positive map radius. This can be done by the built-in method **Lerp**.
- The last step is to place a map pointer in the calculated 2D position.

The map requires to be updated constantly in order to display the current state of the game. That means that the method explained above has to be called for every game entity, and at quite high rate. This is a huge calculation load.
Updating the map every frame would be unnecessary, since so much smoothness is not required. Most of entities are static and the units that are the only ones that move are relatively slow, and updating the map few times a second is enough to display an accurate situation of the game. That is why the map is updated every 0.05 seconds or 20 times per second using a coroutine.

As well as with the nameplates (explained below), the map pointers acquire its colour depending on the relationship between the player and the entity.
Each resource has its own icon.

Figure 4.2: The current minimap pointers for different entities.

4.6 Nameplates

Nameplates are UI elements that display certain information about an entity. They overlay over its assigned entity, and display:

- The name, for all entities.
- An integrity bar. In the case of resource nodes, it shows the amount of remaining resource. For live entities, it shows their health points.
- If the entity can cast, it shows the cast bar, and the progress is updated.

The nameplates are updated constantly in order to be set on top on the units. To do that, the 3D world position is translated to 2D coordinates in the UI. There is a class that runs a coroutine that performs this action for every nameplate.

Each nameplate is assigned to an entity. In order to avoid updating constantly to check if any value has changed, the **values** of the nameplates are updated from the **live entity** or **resource node** class every time a value changes.

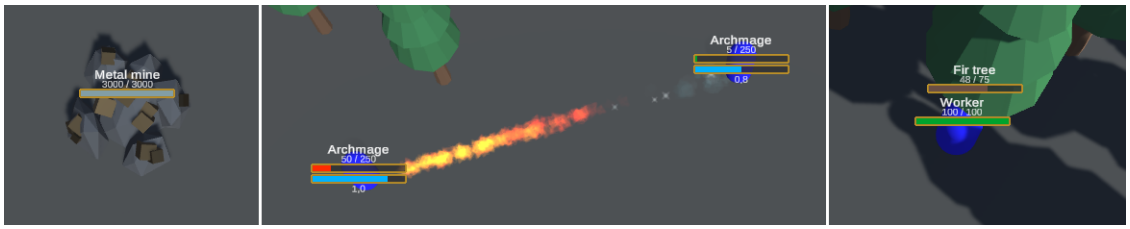

Figure 4.3: Different nameplates displaying information of different entities.

Nameplates also differ depending on the entity.

- If the entity is a live entity, the nameplate will take the colour of the health bar depending on the diplomatic relationship between the player who owns the entity and the player who plays the game.
- If the entity is a resource node, then it will take the colour corresponding to the resource node: brown for trees, light grey for metal, etc.

The nameplates are shown when an entity is selected but also when the player puts the mouse over the entity. That is helpful when the player has a whole formation selected but wants to check the health integrity of the units it is attacking.

Chapter 5: Optimisation decisions

This chapter is focused on explaining several optimization choices and techniques that were not explained before because they did not fit correctly or are too wide concepts, used in several parts.

5.1 Coroutines before updates

Unity method **Update** is not as efficient as it looks. That is why in this game it has only been used once as explained in the first sections. The rest works with coroutines. They are sometimes harder to handle and can lead to bugs if not used properly, but when used correctly they reward with an improved performance allowing the programmer to take advantage of multi-processing.

5.1.1 Optimised coroutine loops with variable rate

Certain loops do not require extensive precision. For example the map, does not need to be updated at each frame. The programmer can choose how much how much it will wait between each loop execution. That is why most of updates have been programmed to be tweaked in order to have a higher or lower rate in order to improve performance.

5.2 Layered colliders

Game objects are split in **layers**. A layer is like a domain, they are given by the Unity engine and can be used in many ways. One way is for the physics. Since a lot of colliders do not interact between them, it is unnecessary to check for their collisions. For example, water would never collide with the ground, or a projectile launched by an entity cannot hit the entity itself.
This way, only the corresponding colliders collide with each other, reducing the calculations significantly.

Layers are not limited to colliders. They can also been used when **Ray casting** or **manual collision check**, for example, with **OverlapSphere**, a Unity built-in method widely used by projectiles and units.

Chapter 6: Unimplemented features

Mainly due to of time restrictions, some features could not be implemented. These features are explained in the following sections.

6.1 Artefacts of power

Artefacts of power were going to be objects that would grant a huge bonus to any player that held them. It was also supposed to give the game its name.
An **artefact** would have be an entity that would need to be carried by a capable unit. Could either be an item, which would be stored in a unit inventory, or an object that should be placed somewhere. Some of the planned artefacts were:

- **Magic nullifier:** It was supposed to be an object that could be taken and placed by a unit. The object would be activated manually by the last player who took it, and when activated, it would create a field were no live entity could use any ability. It would also drain any combat resource from any unit.
- **Weapons:** Certain weapons could deliver extra elemental damage and grant certain spells.
- **Power scrolls:** Power scrolls would grant "**one use spells**" to the unit that held it. Power scrolls should only be carried by **heroes**.
  Most of the planned spells were some kind of area attack that could destroy whole formations or a considerable amount of buildings, like a firestorm or an earthquake.
  Those were supposed to be the most powerful ones.

They have not been implemented because they would require certain balance and they are not that important for a functional game, which was the main purpose of this project.
Additionally, it would require little effort to create a basic, functional version of them.

6.2 Fog of war

Fog of war is a widely used feature in RTS games. It consist in occluding any area that is not visible by any friendly unit, making this way impossible to know what the enemy does outside the player visual range.
This system implies a lot of processing and calculations, since it has to be taken in mind the sight radius of every friendly entity and check for every enemy entity.
This could be implemented in different means.

6.2.1 Mask shader

This may be easier to implement than the occlusion matrix with the correct knowledge about Unity shaders, but not necessarily the optimal. It consists into creating a disc or sphere with the entity position at the center and its visual range as the radius. The polygon, let's call it **visual area**, but act as a mask that interacts with enemy entities.
To do this, the visual area would use a shader, which renders the surface of a model only when both the enemy entity and the visual area **intersect**.

As a downside, a part of overusing the GPU, would be that entities that are in the edge of the visual area would be rendered in half, as shown in the following figure. The colour red symbolises the occluded surface, while the green symbolises the rendered surface and the blue the observer, which will always be shown since it is friendly and there is no need to mask-check it.

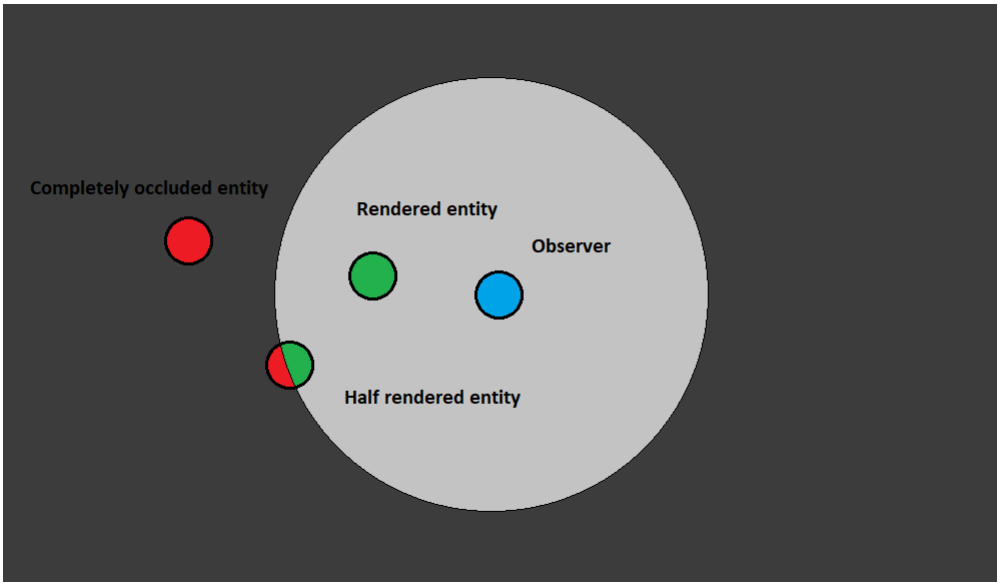

Figure 6.1: A simple scheme showing the problems that come with masking.

6.2.2 Occlusion matrix

On the other hand, it is possible to make a pure CPU based system. It consists into virtually dividing the map by n by m cells. Each cell is a Boolean that can be true or false (visible or out of sight).
Keeping this in mind, this matrix has to be updated every time a unit moves. So, while a unit is moving it is also updating the state of the matrix. If this is approached by using multi-threading, it would require a strong synchronisation system. If not, the performance could be compromised.

It is necessary to calculate, for each cell, if the distance between it and the observer is equal or less than the observer sight radius. The distance check if quite performance-expensive, but it is not necessary to check for every cell in the map. The performance can be improved by only checking the ones that are in the x or y radius. This way, it is possible to add certain limits. For example, if the observer is at (20,20) and has a sight radius of 5, it is not necessary to check a cell that is at (22,100) since the y is completely out of range.
For the cells inside the "square range", it is necessary to measure the distance, but the amount of cells to check has been greatly reduced.

Now it is as simple as not rendering the units that are in occluded cells. The problem is that, since this game does not use a tilled or tessellated map, it requires an extra calculation to define in what cell an entity is. Note that, as said before, cells are not real, but just a reference which takes certain coordinates from a non-discrete map. This means that units can lay in infinite positions (always between the bounds of the map). To calculate the cell means to check the bounds of a entity, simply considering that it lays between four points, which are the bounds or corners of the cell.

As a downside, this system does not create round borders around the sight radius of an entity. The "resolution" of such edge can be improved by increasing the cell matrix density, but that has a negative impact, and the cost increases quadratically.

6.3 Game map and terrain

The terrain right now is just a plane. A map should be done by hand and controlling where the resources are placed. Since this game is supposed to be only a test scene and create the basics of a game of this kind, it has been considered not necessary to add more than what it is in the scene. Creating a real map would only take time, but would not require any programming nor modelling (except for decoration).

Procedural generation has been considered since I have certain background with it. The problem is that procedural generation would not fit a game of this kind since it is chaotic. That means, not all the map is distributed uniform and with this comes also comes the resources. That would lead, not only to defective maps (for example, players completely surrounded by mountains or forests but also unbalanced resources, giving advantage to certain players randomly.

6.4 Path finding

Path finding is by far the most important feature that has not been added to this project. It is necessary not only to avoid hitting obstacles such as buildings or resources but also to keep the units arranged and not trespassing each other.

This feature could be easily added using the built-in Unity method collection of ***Nav meshes****. The idea at the beginning was to create my own system which would do the same but using my own methods and only the necessary calculations. At the end, that turned out to be harder than expected and the project has been left with no path finding at all.*

Bibliography

[1] Sound effects source, https://freesound.org/

[2] Sprites source, https://opengameart.org/

[3] Raycast Unity scripting API, https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

[4] OverlapSphere Unity scripting API, https://docs.unity3d.com/ScriptReference/Physics.OverlapSphere.html

[5] Vector2 Unity scripting API, https://docs.unity3d.com/ScriptReference/Vector2.html

[6] Vector3 Unity scripting API, https://docs.unity3d.com/ScriptReference/Vector3.html

[7] Select units within a rectangle in Unity, https://www.habrador.com/tutorials/select-units-within-rectangle/

[8] Circle-line intersection point, https://stackoverflow.com/questions/300871/best-way-to-find-a-point-on-a-circle-closest-to-a-given-point